NTNU
Norwegian University of
Science and Technology

Decoupling deep learning and reinforcement learning for stable and efficient deep policy gradient algorithms

Alfredo Vicente Clemente

Master of Science in Computer Science

Submission date:	August 2017
Supervisor:	Helge Langseth, IDI
Co-supervisor:	Humberto Castejon, Telenor Research
	Arjun Chandra, Telenor Research

Norwegian University of Science and Technology
Department of Computer Science

Abstract

This thesis explores the exciting new field of deep reinforcement learning (Deep RL). This field combines well known reinforcement learning algorithms with newly developed deep learning algorithms. With Deep RL it is possible to train agents that can perform well in their environment, without the need for prior knowledge. Deep RL agents are able to learn solely by the low level percepts, such as vision and sound, they observe when interacting with the environment.

Combining deep learning and reinforcement learning is not an easy task, and many different methods have been proposed. In this thesis I explore a novel method for combining these two techniques that matches the performance of a state of the art deep reinforcement learning algorithm [Clemente et al., 2017] in the Atari [Mnih et al., 2013] domain for the game of Pong, while requiring fewer samples.

Preface

This is the master's thesis for my master's degree in computer science at the Norwegian University of Science and Technology, with the support of Telenor Research.

I would like to thank my supervisor Helge Langseth for his support. Special thanks to Humberto Castejon and Arjun Chandra for their feedback and motivation, and for leading an excellent research group with Mikkel Sannes Nylend, Per Torgrim F. Thorbjørnsen, Per Magnus Veierland and myself.

Finally I would like to thank my mother Giuseppina Ragone for her support and encouragement throughout this project.

Alfredo Vicente Clemente Ragone
Trondheim, August 19, 2017

Contents

1 Introduction **1**
- 1.1 Motivation 2
- 1.2 Goals and Research Questions 2
- 1.3 Thesis structure 3

2 Background Theory **5**
- 2.1 Reinforcement Learning 5
  - 2.1.1 Value functions 7
  - 2.1.2 Learning an optimal policy 8
  - 2.1.3 Importance Sampling 13
- 2.2 Deep learning 14
  - 2.2.1 Artificial Neural Networks 14
  - 2.2.2 Feedforward Neural Networks 16
  - 2.2.3 Convolutional Neural Networks 17
  - 2.2.4 Learning in Artificial Neural Networks 18
- 2.3 Deep Reinforcement Learning 22
  - 2.3.1 Deep Q Learning 23
  - 2.3.2 Deep Policy Gradient Algorithms 25

3 Related Work **29**

4 Model **33**
- 4.1 Parametrizing the policy 34
- 4.2 Optimizing the Weights of the Neural Network 36
  - 4.2.1 Logit Target 36
  - 4.2.2 Policy Target 36
- 4.3 Proposed Algorithm 38

5 Experiments **41**
5.1 Setup 41
5.2 Implementation 42
5.2.1 Experience Replay 43
5.3 Results 43
5.3.1 PAAC Baseline 46

6 Discussion **49**

7 Conclusion and Future Work **51**

Appendices **53**

A **55**

B **57**

Bibliography **59**

List of Figures

2.1 Reinforcement learning problem 6
2.2 Simple artificial neuron with three inputs and an output 14
2.3 Simple artificial neural network 15
2.4 Three common activations. In order hyperbolic tangent, logistic function and rectified linear unit. 16
2.5 Fully connected feedforward neural network with 4 input units, 4 hidden units and 2 output units. 16
2.6 Convolution of two locations of a volume of size $h \times w \times d$ with a kernel of size $k \times k$. 18
2.7 Over-fitting of a learning algorithm. 19
2.8 Comparison of a single gradient estimate and update, with two gradient estimates and updates. 21
2.9 Illustration of the architecture used by Mnih et al. [2013]. 24

5.1 Sample frame from the game of Pong. 41
5.2 Sample input to the network. 42
5.3 Average performance of three runs with the best found hyperparameters. 44
5.4 Average performance of three runs, using different amount of target reuse. 45
5.5 Performance keeping targets for different amounts of timesteps . . 45
5.6 Performance of the proposed algorithm and PAAC on the game of Pong. 46
5.7 Performance of off-policy PAAC without (left), and with (right) importance sampling, for multiple replay memory sizes.. 47

Chapter 1

Introduction

In the last decade there has been an noticeable increase in the effectiveness of machine learning systems, and more specifically deep learning (DL) systems. Krizhevsky et al. [2012] used a deep learning system to achieve the best top-5 error of 15.3% in image classification for the 2012 Large Scale Visual Recognition Challenge [Russakovsky et al., 2014], surpassing the second place by 10.9%. This event rekindled the machine learning community's interest in neural networks. Deep learning systems now represent the state of the art within image recognition, image segmentation, speech recognition, and natural language parsing.

Recent advances in reinforcement learning (RL) have come from the combination of existing RL techniques with new deep learning techniques to obtain superhuman performance in a multitude of complex, high-dimensional tasks such as Go [Silver et al., 2016], Atari [Mnih et al., 2013] and VizDoom [Lample and Singh Chaplot, 2016].

After the first wave of deep reinforcement learning algorithms appeared, there have been many improvements to their computational efficiency [Mnih et al., 2016], time efficiency [Nair et al., 2015] and sample efficiency [Schaul et al., 2015]. However, the environments in which these techniques can be applied is still very limited.

In order to solve more complex problems with deep reinforcement learning, such as self-driving cars, conversational systems and high level problem solving, these algorithms must significantly improve their sample efficiency. State of the art models, even in simple domains [Wang et al. [2016], Clemente et al. [2017], Pritzel et al. [2017]], require up to hundreds of millions of interactions with the environment.

This thesis gives a thorough introduction into reinforcement learning, deep learning and the state of the art within deep reinforcement learning. Finally, a novel method of combining deep learning and reinforcement learning is proposed and compared to a state of the art deep reinforcement learning algorithm in the Atari domain.

1.1 Motivation

Current developments in deep reinforcement learning algorithms have lead to increased sample efficiency, better performance, and reduced training time. Sample efficiency and performance can be improved by storing experiences in a replay memory [Mnih et al. 2013; Wang et al. 2016; Pritzel et al. 2017] and then using them for off-policy updates [Mnih et al. 2013; Schaul et al. 2015; Wang et al. 2016] or for other learning tasks [Jaderberg et al., 2016]. Training time is reduced by either decreasing the time required to learn from each sample [Clemente et al., 2017] or simply reducing the amount of samples required.
As deep rl progresses, we wish to apply it to more complex environments and tasks, however it is clear that sample efficiency and training time must be greatly reduced. I propose an innovative method for combining reinforcement learning and deep learning to create more sample efficient deep reinforcement learning algorithms.

1.2 Goals and Research Questions

This thesis has two main goals and one research question

Goal 1

Understand the state of the art in deep reinforcement learning.

Deep reinforcement learning is a combination of many fields of research, including statistics, mathematics, and physics among others. To reason about how and why these state of the algorithms work, it is essential to have a good understanding of these fields. With this knowledge in hand, an intuitive understanding of the plethora of deep reinforcement learning algorithms can be achieved. Allowing for the development of novel algorithms.

Goal 1

Improve the sample efficiency of deep reinforcement learning algorithms.

Current reinforcement learning algorithms require hundreds of millions of interactions with their environments to learn. If we wish to apply these algorithm to real world problems, the amount of environment interactions must be vastly reduced. This leads tot he main research question of this thesis.

Research question

Why do state of the art deep reinforcement learning algorithms require such a large amount of environment interactions, and how can this be reduced.

To answer this question it is vital to gain a deep understanding of deep reinforcement learning algorithms, and find common factors across them.

1.3 Thesis structure

Chapter 2 provides the information required to understand the problem at hand. Section 2.1 introduces reinforcement learning, Section 2.2 introduces deep learning and Section 2.3 describes what deep reinforcement learning is and the current state of the art in the field.
Chapter 3 presents similar approaches.
Chapter 4 presents the proposed algorithm.
Chapter 5 contains the experiments performed to validate the proposed algorithm.
Chapter 6 discusses the results of the experiments and their importance.
And finally chapter 7 concludes the thesis and provides some avenues for further research.

Chapter 2

Background Theory

Deep reinforcement learning is a newly born field of machine learning that arose from the combination of decades of research in the field of reinforcement learning, and the current advances within deep learning. I will first introduce reinforcement learning, starting from the basics and work towards current techniques. I will then introduce deep learning and its importance to reinforcement learning, and finally I will show how we can, and why we would, combine the fields of reinforcement learning and deep learning into deep reinforcement learning.

2.1 Reinforcement Learning

The goal of reinforcement learning is to create agents that by interacting with an environment are able to learn a behaviour that maximizes its performance.
This system is formally modelled as a finite Markov Decision Process (MDP). This MDP is defined with a 3-tuple $(S, A, p(s', r|s, a))$ where S is the set of all possible states, A is the set of all available actions and $p(s', r|s, a)$ is the probability of transitioning to state s' and receiving rewards r when being in state s and taking action a. An agents in this model observes the current state $s_t \in S$ of the environment at time t and chooses an action $a_t \in A$ according to its policy π. The actor's policy is a function that maps the environment's state to actions probabilities such that $\pi(a_t|s_t)$ is the probability of taking action a_t. The chosen action is then applied to the environment, which in turn generates a new state s_{t+1} and a scalar reward r_{t+1}. The agent-environment interaction is shown in Figure 2.1.

The performance of an agent is measured by the expected sum of future discounted rewards given the current state s_t and following the current policy π.

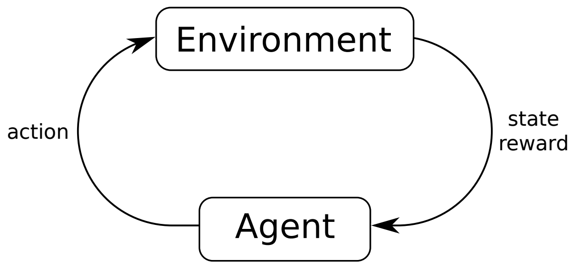

Figure 2.1: Reinforcement learning problem

This is referred to as the return, and is defined as

$$R_t = \sum_{a'} \pi(a'|s_t) \sum_{(s',r')} r'p(s',r'|a_t,s_t) + \gamma R_{t+1} \tag{2.1}$$

$$R_t = \mathbb{E}_\pi\Big[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}\Big] \tag{2.2}$$

To motivate the usage of a discount factor, we may consider an alternative definition of return:

$$R'_t = \mathbb{E}_\pi\Big[\sum_{k=0}^{T} r_{t+k+1}\Big]$$

where T is the final time-step, at which the the environment is at its terminal state. However, not all environments have a terminal state, some may continue for eternity and will therefore result in R'_t being unbounded. In order to optimize the policy with regard to the return, the return must be bounded. The addition of a discount solves this issue by giving an upper bound on R_t:

$$\sum_{k=0}^{\infty} \gamma^k = \frac{\gamma}{1-\gamma}$$

$$R_t = \mathbb{E}_\pi\Big[\sum_{k=0}^{\infty} \gamma^k \cdot r_{t+k+1}\Big] \leq \sum_{k=0}^{\infty} \gamma^k \cdot r_{\max}$$

$$R_t \leq \frac{\gamma}{1-\gamma} \cdot r_{\max} \tag{2.3}$$

where $r_{\max}$ is the maximum obtainable reward in any given state.

The goal of a reinforcement learning agent is to learn an optimal policy, that maximizes the expected return $\eta(\pi)$, abbreviated η_π, for all possible starting states given by

$$\eta_\pi = \mathbb{E}_{s_0}[R_0] \tag{2.4}$$

This policy is called an optimal policy and is defined as

$$\pi_* = \arg\max_{\pi'} \eta_{\pi'} \tag{2.5}$$

We can now use these definitions to introduce the concept of value functions.

2.1.1 Value functions

It is often helpful in the context of reinforcement learning, to have some estimate of how valuable it is for an agent to be in a state. A state-value function, often referred to as a value function, maps states to the expected return by following a given policy policy π. In other words, it represents how valuable it is for an agent to be in a given state, where value is defined in terms of return. The rewards an agent receives are directly affected by its policy, therefore it only makes sense to speak of the value of a state given a specific policy. Formally a value function is defined as:

$$v(s_t|\pi) = R_t = \mathbb{E}_\pi[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots |s_t] \tag{2.6}$$

We can devise a recursive formulation for v by observing the value of the next state

$$v(s_{t+1}|\pi) = \mathbb{E}_\pi[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots |s_{t+1}] \tag{2.7}$$

we can then combine Equation (2.6) and Equation (2.7) to present the Bellman equation

$$v(s_t|\pi) = \mathbb{E}_\pi[r_t + \gamma v(s_{t+1}|\pi)|s_t] \tag{2.8}$$

We can similarly define a state-action-value function, referred to as a Q function, that maps state action pairs to expected return. It represents the return an agent obtains in expectation by taking action a_t while being in state s_t, and then following the policy π for all future time-steps:

$$q(s_t, a_t|\pi) = \mathbb{E}_\pi[r_t + \gamma v(s_{t+1}|\pi)|s_t, a_t] \tag{2.9}$$

2.1.2 Learning an optimal policy

There are many algorithms designed to learn optimal policies, however I will limit the scope of the algorithms I present to the families of Temporal Difference (TD) methods and Policy Gradient methods for control.

Temporal Difference Methods for Control

Temporal difference methods belong to the family of value-based reinforcement learning algorithms, which learn a state-action-value function $Q(s_t, a_t) \approx q(s_t, a_t)$ by interacting with the environment using some policy π and observing some experience. TD methods update their Q function to give better return estimates for the current state and action. With a Q function available, it is possible to produce a policy by examining all available actions for the current state and choosing the one that generates the maximum accumulated future reward $\pi(a_t|s_t) = \arg\max_{a' \in A_t} Q(s_t, a_t)$.

Intuitively, TD methods learn by updating the Q function as to reduce the error given by $R_t - Q(s_t, a_t)$. Given that R_t is an expectation over all possible rewards and states by following the policy π, it is often not possible to compute it for non-trivial problems. Calculating the return for a simple environment with 5 possible states and 5 possible actions that always terminates after 100 timesteps would take time in the order of $(5 \cdot 5)^{100}$, more than there are atoms in the known universe. Additionally $p(s', r|s, a)$ may not be known. The most intuitive way of estimating R_t is by simply interacting with the environment and storing all rewards. Once the episode is over, the return $\tilde{R}_t$ for that episode can be calculated. $\tilde{R}_t$ is an unbiased estimator of R_t, that does not require knowing the probability distribution of states and rewards. With this approach however, the agent must reach the a terminal state of the environment, which may require a long time or may not be possible at all. The TD methods I will present estimate R_t by bootstrapping from a combination of the values of the current Q function and observed rewards.

The simplest form of TD learning is an on-policy algorithm called Sarsa [Sutton and Barto, 1998]. In this algorithm the agent observes the current state of the environment s_t, and generates an action a_t following its behaviour policy $\pi(a_t|s_t) = f(A_t, s_t, Q)$ for some function f. Generally f is chosen to have some stochasticity to ensure that things learned early in training do not limit the portions of the state and action space that are explored. The agent learns by taking an action a_t in its environment and observing the next state s_{t+1} and reward r_{t+1}. Then a new action a_{t+1} is chosen from the same policy. In Sarsa R_t is estimated with the one-step return given by $\tilde{R}_t^{(1)} = r_{t+i} + \gamma Q(s_{t+1}, a_{t+1})$. The

Q function in then updated using this estimate

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\big[R_t - Q(s_t, a_t)\big] \tag{2.10}$$

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\big[r_{t+i} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\big] \tag{2.11}$$

The off-policy version of Sarsa is called Q-learning and follows the same procedure to generate actions. However, it estimates $\tilde{R}_t^{(1)} = r_{t+i} + \gamma \max_{a'} Q(s_{t+1}, a')$ and applies the update

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\big[r_{t+i} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)\big] \tag{2.12}$$

where α the learning rate, hyperparameter that regulates how fast the Q function changes.

The slight difference in the estimates of the return between Sarsa and Q-learning illustrates the contrast between on-policy and off-policy learning. In on-policy learning the estimate of future rewards is done assuming future actions will be drawn from the current policy, while off-policy learning assumes that future actions are chosen from the greedy policy $\pi(a_t|s_t) = \arg\max_{a' \in A_t} Q(s_t, a')$. Q-learning and Sarsa are equivalent when the behaviour policy is the greedy policy.

One-step estimates of the return bootstrap from the Q value of other state-action pairs while grounding itself a single time step of reality, in the form of the reward observed. Bootstrapping can be beneficial because it allows algorithms to update their estimates on-line without having to observe the full return.

One-step methods are a special case of the general class of n-step methods in which the estimate of the return is grounded on one or more time-steps

$$\tilde{R}_t^{(n)} \doteq r_t + \gamma r_{t+1} + \ldots + \gamma^{n-1} Q(s_n, a_n) \tag{2.13}$$

n-step methods offer a parameter n to determine the bias-variance trade-off that is commonly encountered in machine learning. Higher values of n will give a better estimate of the true return and therefore reduce the bias. On the other hand lower values of n will have lower variance. In practice the optimal value of n depends on the problem and resides in the range $n \in [1, T]$. At $n = T$ we have Monte Carlo learning, which does no bootstrapping and simply calculates the return with all observed rewards once the episode is over.

The simplest implementation of Q learning or Sarsa is with a lookup table. The table maintains an entry for each state-action pair, which contains the Q value of that pair. The Q values are then updated with Equation 2.10 or 2.12. This approach however is not well suited for problems with large, or even continuous, state or action spaces. As we shall later see, these approaches can be combined with powerful function approximators to address this issue.

Policy Gradient Methods

Policy gradient methods are fundamentally different from the value based methods presented previously. Instead of estimating a value function and deriving a policy from it, policy gradient methods directly learn a parametrized policy function $\pi(a_t, s_t; \theta)$ for some parameters θ. Although policy gradient methods do not use value functions to choose actions, they may still make use of them to learn the policy.
Theoretically the only requirement for the function that parametrizes the policy is that it is differentiable with respect to its parameters. However in practice it is generally required that the policy always assigns some non-zero probability to all available actions on all states to ensure exploration.

The goal with these methods, as with all reinforcement learning methods, is to devise a policy that maximizes the expected return η. The policy gradient theorem [Sutton et al., 1999] provides a way of directly calculating the gradient of η_{π_θ} with respect to the policy parameters θ.

$$\nabla_\theta \eta_{\pi_\theta} = \sum_{s' \in S_t} \sum_{t=0}^{\infty} \gamma^t p(s_t = s' | s_0, t, \pi) \sum_{a' \in A_t} q(s', a' | \pi) \nabla_\theta \pi(a' | s' \theta) \tag{2.14}$$

where $p(s_t = s' | s_0, t, \pi)$ is the probability of s' being the state at time step t given an initial state s_0 by following the policy π.

The simplest of policy gradient methods called REINFORCE [Williams, 1992] approximates Equation (2.14) by using sample estimates. This adaptation is done by first replacing the probability of current state by the expectation of sampled states given the current policy

$$\nabla_\theta \eta_{\pi_\theta} = \mathbb{E}_{s_t \sim \pi_\theta} \big[\gamma^t \sum_{a' \in A_t} q(s_t, a' | \pi) \nabla_\theta \pi(a' | s_t, \theta) \big] \tag{2.15}$$

then similarly the sum over actions is replaced by the expectancy of the chosen action a_t given the current policy

$$\nabla_\theta \eta_{\pi_\theta} = \mathbb{E}_{s_t \sim \pi_\theta}\big[\gamma^t \sum_{a' \in A_t} \pi(a'|s_t,\theta) q(s_t,a'|\pi) \frac{\nabla_\theta \pi(a'|s_t,\theta)}{\pi(a'|s_t,\theta)}\big] \tag{2.16}$$

$$\nabla_\theta \eta_{\pi_\theta} = \mathbb{E}_{(s_t,a_t) \sim \pi_\theta}\big[\gamma^t q(s_t,a_t|\pi) \frac{\nabla_\theta \pi(a_t|s_t,\theta)}{\pi(a_t|s_t,\theta)}\big] \tag{2.17}$$

finally using Equation (2.9) we can replace q with the return

$$\nabla_\theta \eta_{\pi_\theta} = \mathbb{E}_{(s_t,a_t) \sim \pi_\theta}\big[\gamma^t R_t \frac{\nabla_\theta \pi(a_t|s_t,\theta)}{\pi(a_t|s_t,\theta)}\big] \tag{2.18}$$

Given that $\nabla_\theta \log \pi(a_t|s_t,\theta) = \frac{\nabla_\theta \pi(a_t|s_t,\theta)}{\pi(a_t|s_t,\theta)}$ we can rewrite the above as

$$\nabla_\theta \eta_{\pi_\theta} = \mathbb{E}_{(s_t,a_t) \sim \pi_\theta}\big[\gamma^t R_t \nabla_\theta \log \pi(a_t|s_t,\theta)\big] \tag{2.19}$$

The true gradient can now be estimated using the Monte Carlo estimate of the return $\tilde{R}_t$, which results in a way of calculating the gradient of the metric η we are trying to optimize with respect to the parameters θ that affect it. REINFORCE uses this gradient to perform gradient ascent on the policy weights with the update

$$\theta_{i+1} \leftarrow \theta_i + \alpha \gamma^t \tilde{R}_t \nabla_\theta \log \pi(a_t|s_t,\theta) \tag{2.20}$$

Actor-critic algorithms estimate the return R_t with a learned Q function $Q(s_t,a_t|\theta_Q) \approx q(s_t,a_t|\pi)$, so that

$$\nabla_\theta \eta_{\pi_\theta} = \mathbb{E}_{(s_t,a_t) \sim \pi_\theta}\big[\gamma^t Q(s_t,a_t|\theta_Q) \nabla_\theta \log \pi(a_t|s_t,\theta)\big] \tag{2.21}$$

The policy gradient theorem can be extended to include a baseline $b(s_t)$ [Williams, 1992] that is a function of the state

$$\nabla_\theta \eta_{\pi_\theta} = \sum_{s' \in S_t} \sum_{t=0}^{\infty} \gamma^t p(s_t = s'|s_0,t,\pi) \sum_{a' \in A_t} (q(s',a'|\pi) - b(s')) \nabla_\theta \pi(a'|s'\theta) \tag{2.22}$$

The addition of this baseline does not affect the expectation of $\nabla\eta$, but can reduce the variance of the updates [Sutton and Barto, 1998]. To illustrate the usefulness of a baseline function we consider an environment in which the true value of a state is given by $v(s) = 1000 + \phi(s)$ for $\phi(s) \in (-1,1)$. The value of a state-action pair $q(s,a)$ will then be in the range $(999, 1001)$, meaning that the value of best and worst possible actions can only differ by upto 0.2%. Choosing $b(s) = 1000$ will in this case result in $[q(s,a) - b(s)] \in (-1,1)$ where the relative difference in value of the best and worst action is upto 100%. This has been shown to significantly increase learning performance [Sutton and Barto, 1998].

Advantage actor-critic algorithms are on-policy policy gradient algorithms similar to REINFORCE, that use a specially chosen baseline function so that $b(s) = v(s)$. Given this choice of a baseline function, we can interpret the difference $q(s,a) - b(s)$ as the advantage function, which can be defined as

$$adv(s,a) = q(s,a) - v(s) \tag{2.23}$$

and represents the value lost by choosing an action that is not the optimal action [Baird III, 1993].
We can then follow the same steps used to derive (2.19) with (2.22) as the starting point, resulting in the update

$$\theta_{i+1} \leftarrow \theta_i + \alpha\gamma^t [R_t - b(s_t)] \nabla_\theta \log \pi(a_t|s_t, \theta) \tag{2.24}$$

In addition to this choice of the value function as a baseline, advantage actor-critic algorithms estimate the return R_t by bootstrapping from current estimates of the very same value function $V(s;\theta_v) \doteq v(s)$ used as a baseline, similar to TD methods. Using the one-step estimate $\tilde{R}_t^{(1)} \doteq G_t$ would yield the update for the policy parameters

$$\theta_{i+1} \leftarrow \theta_i + \alpha\gamma^t [\tilde{R}_t^{(1)} - V(s_t;\theta_v)] \nabla_\theta \log \pi(a_t|s_t, \theta) \tag{2.25}$$
$$\theta_{i+1} \leftarrow \theta_i + \alpha\gamma^t [r_t + \gamma V(s_{t+1};\theta_v) - V(s_t;\theta_v)] \nabla_\theta \log \pi(a_t|s_t, \theta) \tag{2.26}$$

The value function parameters can be also updated using $R_t^{(1)}$ as its target with the update given by

$$\theta_{v,j+1} \leftarrow \theta_{v,j} + \alpha[(r_t + \gamma V(s_{t+1};\theta_{v,j}) - V(s_t;\theta_{v,j})] \tag{2.27}$$

Advantage actor-critic methods can be extended, similarly to TD methods, to use different estimates of the return, for example the n-step return.

An off-policy version of the policy gradient theorem was presented by Degris et al. [2012] and is given by

$$\nabla_\theta \eta_{\pi_\theta} = \mathbb{E}_{(s_t,a_t)\sim\beta} \left[\frac{\pi(a_t|s_t,\theta)}{\beta(a_t|s_t)} R_t \nabla_\theta \log \pi(a_t|s_t,\theta) \right]. \tag{2.28}$$

This estimates allows the usage of experiences sampled from a behavioural distribution β to improve the current policy π_θ, opening the door for off-policy policy algorithms.

2.1.3 Importance Sampling

While importance sampling is a general technique not limited to reinforcement learning, it is vital to understand off-policy reinforcement learning algorithms.

Importance sampling is a technique used to estimate an expectation under one distribution from samples from a different distribution. Suppose we wish to find the expectation of a function $\tilde{x} = \mathbb{E}_{x \sim \chi}[f(x)]$ with samples of x gathered from the distribution Ω. The expectation can be written as the integral

$$\tilde{x} = \mathbb{E}_{x \sim \chi}[f(x)] = \int f(x) p(x|x \sim \chi) dx \tag{2.29}$$

$$\tilde{x} = \int f(x) p(x|x \sim \chi) \frac{p(x|x \sim \Omega)}{p(x|x \sim \Omega)} dx \tag{2.30}$$

$$\tilde{x} = \int f(x) p(x|x \sim \Omega) \frac{p(x|x \sim \chi)}{p(x|x \sim \Omega)} dx \tag{2.31}$$

$$\tag{2.32}$$

The integral can then be transformed back to the expectation

$$\tilde{x} = \mathbb{E}_{x \sim \Omega} \left[f(x) \frac{p(x|x \sim \chi)}{p(x|x \sim \Omega)} \right] \tag{2.33}$$

where the term $\frac{p(x|x \sim \chi)}{p(x|x \sim \Omega)}$ is referred to as the importance sampling ratio. Given that both $p(x|x \sim \chi)$ and $p(x|x \sim \Omega)$ fall in the range $[0, 1]$, the importance sampling ratio falls in the range $[0, \infty)$.

The expectation can then be estimated from the samples of x with

$$\tilde{x} = \mathbb{E}_{x \sim \Omega} \left[f(x) \frac{p(x|x \sim \chi)}{p(x|x \sim \Omega)} \right] \approx \frac{1}{N} \sum_{i=1}^{N} f(x_i) \frac{p(x_i|x_i \sim \chi)}{p(x_i|x_i \sim \Omega)} \tag{2.34}$$

In reinforcement learning algorithms, importance sampling is used to estimate expectations under the current policy, with samples gathered by executing a different policy. Given that the importance sampling ratio can be infinitely large, the use of importance sampling can lead to infinitely large updates causing instability or even catastrophic divergence under training. Due to this, special techniques must be used alongside importance sampling to achieve stable learning.

2.2 Deep learning

Deep learning is a sub-field of machine learning that is concerned with function approximation through the learning of meaningful hierarchical representations of data. It involves techniques from the fields of mathematics, statistics, physics, psychology and neuroscience. Function approximation is of vital importance for reinforcement learning. Consider the game of chess, this game has approximately 10^{47} distinct states. If we were to represent the value of each state with a single bit we would require approximately 10^{36} GB of memory. To put this number into perspective, the combined storage capacity currently available in the world is under 10^{13} GB. Function approximation allows us to approximate value and policy functions for a reinforcement learning problem using a number of parameters much smaller than the amount of states in the problem.

2.2.1 Artificial Neural Networks

All deep learning models are some variation of artificial neural networks (ANN). ANNs are computational models loosely inspired by a biological brain. The goal of an artificial neural network is to approximate some function $f^*(x_i)$ that maps the input $x_i \in X$ for the dataset $X \sim \mathbb{X}$ to their corresponding target $y_i = f^*(x_i) \sim \mathbb{Y}$. This is done by learning a parametrized function $f(x;\theta) \approx f^*$ with parameters θ. In the case where $\mathbb{Y}$ is a finite set this is a task of classification, and when $\mathbb{Y}$ is not finite it is a task of regression.

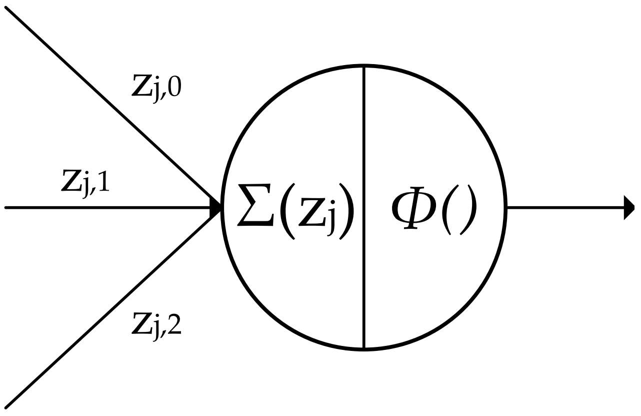

Figure 2.2: Simple artificial neuron with three inputs and an output

ANNs are composed of a large collection of simple processing units, referred to as artificial neurons, which receive some inputs and then perform some computation on them to produce an output. Figure 2.2 shows a simple neuron which takes 3 inputs and applies a function to them to produce an output.

There are a set of special neurons referred to as input units which receive inputs to the ANN, output units in turn are connected to the output of the ANN, all other neurons are hidden units. A depiction of an ANN is presented in Figure 2.3.

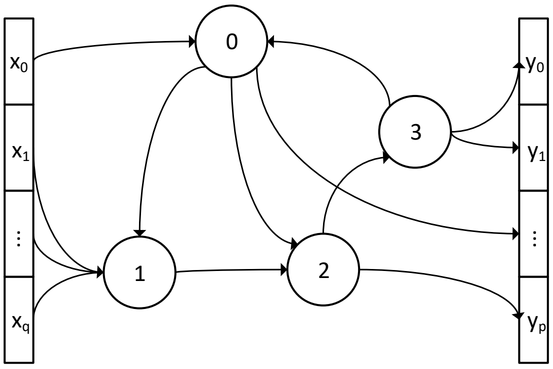

Figure 2.3: Simple artificial neural network

Moving away from the biological view, we can regard each neuron j as a processing unit that produces the output

$$o_j = \phi_j(w_j^T z_j + b_j) \tag{2.35}$$

by performing a linear combination of its input vector z_j with the learned weight vector w_j adding a bias b_j, and then applying the activation function ϕ_j. Common activation functions include the hyperbolic tangent function which maps real values to the range $(-1, 1)$, the logistic function $\phi(z) = \frac{1}{1+e^{-z}}$ maps to the range $(0, 1)$, and the rectified linear unit (ReLU) [Nair and Hinton, 2010] $\phi(z) = \max(0, z)$ maps to the range $[0, +\infty)$. These three activation functions are shown in Figure 2.4.

ANNs can be connected in arbitrary ways, so that any neuron can be an input, output and/or hidden unit, and can be arbitrarily composed to that the output of a neuron can be the input to an other neuron or itself. Different connection schemes will afford networks with different properties and trade-offs, in the following sections I will present two of the most widely used.

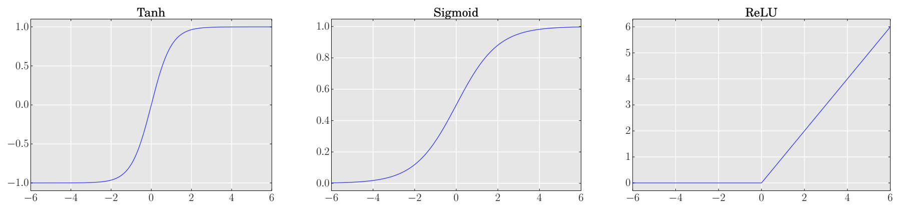

Figure 2.4: Three common activations. In order hyperbolic tangent, logistic function and rectified linear unit.

2.2.2 Feedforward Neural Networks

Feedforward neural networks (FFN) are a constrained type of ANN in which neurons are organized into layers, and each layer receives inputs from the previous layer and produces outputs, that in turn are the inputs for following layers, in the direction from input to output. Figure 2.5 shows a simple FNN.

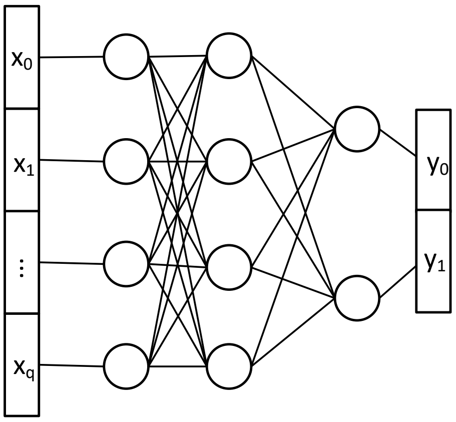

Figure 2.5: Fully connected feedforward neural network with 4 input units, 4 hidden units and 2 output units.

The most basic of feedforward neural networks is the fully connected FFN, in which all neurons in a layer receive the output of all neurons in the previous layer as an input. The first layer is composed of input units, the final layer is composed of output units and all layers in between are composed of hidden units.
Each layer L_l of the network is simply a set of neurons. It is common for all

neurons in a layer L_l to use the same activation function, so that $\phi_i = \phi_l \; \forall i \in L_l$, additionally all neurons in a layer usually receive the same input $z_i = z_l \; \forall i \in L_l$. Each layer can then be thought of as implementing a function $h_l(z_l; \theta_l)$ where

$$h_l(z_l; \theta_l) = \phi_l(W_l^T z_l + B_l) \tag{2.36}$$

where $W_l \in \mathbb{R}^{|z_l|,|L_l|}$ is a matrix in which the columns are the weight vectors of its neurons, and the elements of the vector $B_l \in \mathbb{R}^{|L_l|}$ are the biases of its neurons, for $W_l, B_l \in \theta_l$. The function ϕ_l is applied element-wise to the resulting vector.

The FFN as a whole implements the function $f(x; \theta) = h_L(...h_1(h_0(x; \theta_0); \theta_1)...; \theta_L)$ as a composition of all layers in the network. The different layers in a FNN can be thought of as representing the input at different levels of abstraction, starting from the raw input and gradually creating higher abstractions of it.

2.2.3 Convolutional Neural Networks

Convolutional neural networks (CNN) are a type of feedforward neural network, that excel at processing data in which adjacent points within a data sample are correlated. For example, in an image adjacent pixels are strongly correlated. Similarly in a video, there is correlation between frames in the time dimension, in addition to the spatial correlation present in images.
CNNs are defined to be neural networks that use a convolution operation, instead of matrix multiplication in at least one of their layers [Goodfellow et al., 2016]. When working with a discrete space, the two-dimensional convolution between the tensors I and K at point i, j is defined as

$$(I * K)(i, j) = \sum_m \sum_n I(i - m, j - n)K(m, n) \tag{2.37}$$

In the case of a CNN, I could be a three dimensional tensor containing image data, and K the three dimensional tensor containing some learned weights, referred to as the kernel or filter. In Figure 2.6 the input I is convolved with the kernel K_0 of size $k \times k$ by computing the sum of the element-wise product of the first $k \times k \times k$ section of the input image with the kernel, then the second one, and so on, producing an output matrix of size $(h - k + 1) \times (w - k + 1)$. Kernels are assumed to have the same depth as their input, so it usually omitted when describing the dimensions of a kernel.

A convolutional neuron can be reasoned about as a standard feedforward layer in which each neuron is only connected to a subset of the layer's input, and all neurons in the layer share the same set of weights. From this view it is clear that CNNs require much fewer parameters than a fully connected FNN, at the

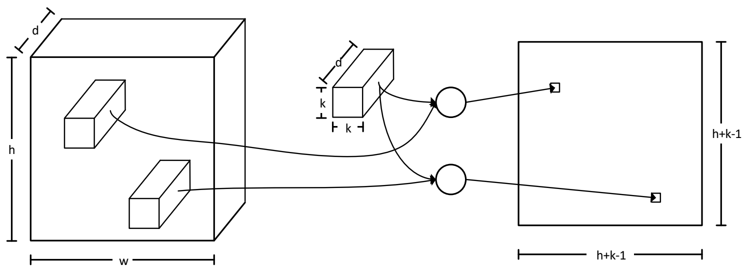

Figure 2.6: Convolution of two locations of a volume of size $h \times w \times d$ with a kernel of size $k \times k$.

expense of reduced expressive power. When the assumption of correlation between adjacent points holds, CNNs can outperform a fully connected FNN while requiring orders of magnitude fewer parameters. A fully connected layer with m inputs and $m - k + 1$ outputs requires $m \times (m - k + 1)$ weights, while a convolutional layer of the same dimensions requires only k weights for a kernel of size k.

Convolutional layers are usually composed of multiple kernels, and their output matrices are concatenated to produce a tensor. The output tensor of a layer can then be used as the input for the next layer, resulting in a hierarchy of layers. It has been shown that the first few layers of a CNN usually learn to detect edges and similar low level features, the following layers then learn to detect meaningful combinations of edges, like corners, deeper layers learn higher level features, for example human faces, or cats.

2.2.4 Learning in Artificial Neural Networks

Learning in the setting of ANNs is done by modifying the set of weights θ of the network in order to improve its performance on a given set of data $S = \{(x_0, y_0), (x_1, y_1), ...(x_n, y_n)\}$ generated from some distribution $\mathbb{S}$. The performance we are trying to improve is usually defined to be the minimization of the true cost

$$L^*(\theta) = \mathbb{E}_{(X,Y)\sim\mathbb{S}}\Big[l(f(X;\theta), y)\Big] \tag{2.38}$$

which is a metric of how different f is from f^* on the true distribution of the data $\mathbb{S}$. In general it is difficult, or even intractable to compute the full expectation in Equation (2.38), so the true cost is approximated with the empirical risk given by

$$L(\theta) = \frac{1}{|S|} \sum_{(x,y)\in S} l(f(x;\theta), y) \tag{2.39}$$

The difference between the true cost and the empirical risk is referred to as the generalization error. Given that calculating the true cost is usually not possible in practice, it is approximated as the empirical risk using a test set $S_{test} \sim \mathbb{S}$ for $S_{test} \cap S = \emptyset$. The test sect acts as a proxy for the true distribution $\mathbb{S}$, and should not be used to train. The case in which an algorithm has a high generalization error is referred to as over-fitting, shown in Figure 2.7, and can for example arise when a model has more parameters that the size of the dataset it is trained on.

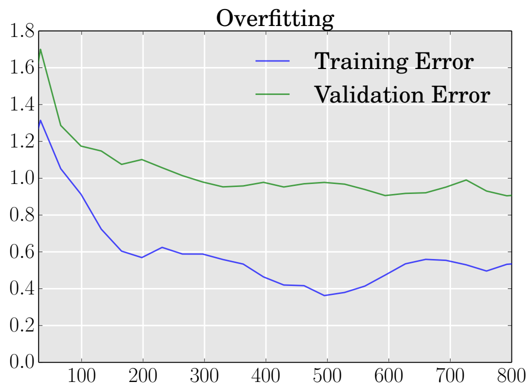

Figure 2.7: Over-fitting of a learning algorithm.

Stochastic Gradient Descent

The task of minimizing a loss function relative to some parameters, is by definition one of optimization. Currently the most common optimization algorithms used to train neural networks are some variant of stochastic gradient descent, or SGD. SGD algorithms iteratively update the model parameters θ in the direction, in parameter space, that most quickly reduces the empirical risk L. This direction is given by the negative of the gradient of the loss with respect to the model parameters, where

$$\nabla_\theta L(\theta) = \frac{1}{|S|} \sum_{(x,y)\in S} \nabla_\theta l(f(x;\theta), y) \tag{2.40}$$

The model parameters are then modified using the update

$$\theta_{i+1} \leftarrow \theta_i - \alpha \nabla_{\theta_i} L(\theta_i), \tag{2.41}$$

where $\alpha \in (0, +\infty)$ is a parameter referred to as the learning rate, used to scale the magnitude of the gradient. In order to guarantee the convergence of SGD the value of α must be reduced throughout the optimization procedure by following some schedule so that $\sum_k^\infty \alpha_k = \infty$ and $\sum_k^\infty \alpha_k^2 < \infty$.

This variant of SGD is usually referred to as batch SGD. The empirical risk and the necessary gradient can be calculated in $O(S)$ time. As the size of current datasets approach the billions samples, it becomes very inefficient to pay such a high computational cost for each parameter update. Due to this and several other reasons that will be discussed shortly, batch SGD is rarely used in practice. In general, deep learning methods tend to be optimized using minibatch SGD, often referred to simply as SGD. In SGD the empirical risk is estimated by using a minibatch S' of samples drawn uniformly from S with

$$\nabla_\theta L(\theta) = \frac{1}{|S'|} \sum_{(x,y)\in S'} \nabla_\theta l(f(x;\theta), y) \tag{2.42}$$

The size of S' is usually in the range between 1 and a few hundred and is independent of S, resulting in a computational cost for estimating the loss in the order of $O(S') \ll O(S)$.

The choice of batch size $n = |S'|$ can have profound effects on the performance of the learning algorithm. Increasing n by a factor of k, reduces the variance of the estimate of the gradient proportional to $\frac{1}{k}$ [Bottou et al., 2016]. Having a better estimate of the direction of the gradient allows the algorithm to take larger steps in that direction by increasing the learning rate, however this step size is limited by some constant dependant on the continuity of f [Bottou et al., 2016]. This increase in batch size will increase the time required to calculate the loss by a factor of k'. Intuitively, given that our estimate of the gradient is limited to be a line, regardless of how good our estimate of the gradient is at some point, the estimated gradient is only valid in locations relatively close to our point of evaluation. Figure 2.8 shows the difference between computing the estimate of the gradient at a single point with high accuracy and taking a large step, compared to taking two smaller steps by computing more inaccurate estimates of the gradient twice.

It is therefore more computationally efficient to calculate multiple noisy estimates of the gradient than calculating the gradient exactly at one point [Wilson and Martinez, 2003], this is illustrated in Figure 2.8.

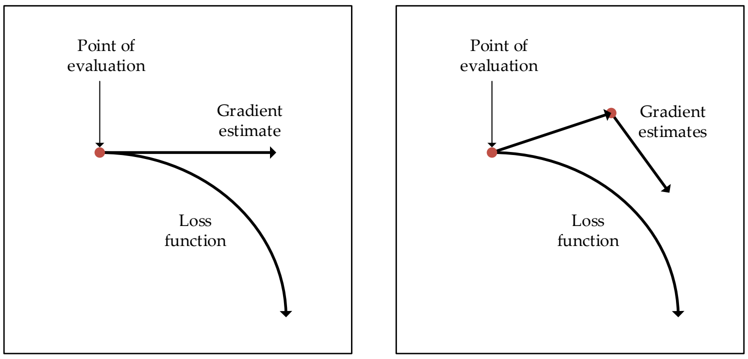

Figure 2.8: Comparison of a single gradient estimate and update, with two gradient estimates and updates.

Thanks to the parallel capabilities of modern compute hardware the time required to calculate the loss for a given batch size usually scales sub-linearly with batch size, meaning that $k' \leq k$. The choice of n and α leads to a trade-off between computational efficiently and sample efficiency in which we attempt to maximize both by choosing n and α to simultaneously maximize $\frac{k}{k'}$ and minimize L. These hyper-parameters are usually found empirically by experimenting with multiple values, either using random search, grid search, or heuristic search.

Loss Functions

The loss function used to measure the quality of f directly defines the objective of the network. For example for a network concerned with identifying whether a picture contains a cat or not one might devise a loss function that gives a loss of 0 for a correct identification and 1 for an incorrect identification (0-1 loss); for a network designed to estimate the current temperature given a history of temperatures an intuitive loss function may be the difference in the predicted temperature and the actual temperature.
However not all loss functions have useful derivatives, for example the 0-1 loss

has a gradient of either zero or undefined at all points. It is not always straightforward to design a loss function that correctly captures the desired dynamics and simultaneously provides meaningful gradients for learning, there are however a few that are widely used.

For regression tasks it is common to use the mean squared error (MSE) given by

$$\mathrm{MSE}(x, y) = \frac{1}{N} \sum_{n} (f(x_n; \theta) - y_n)^2 \tag{2.43}$$

For classification tasks it is common to use the cross-entropy

$$\mathbb{H}(y, x) = - \sum_{n} y_n \log(f(x_n; \theta)). \tag{2.44}$$

2.3 Deep Reinforcement Learning

The human brain is a finely tuned learning machine, we are able to interpret very high-dimensional observations such as images and sounds, learn from very sparse reward signals, and perform calculated and precise actions in the world. It has been shown that human learning can be modelled as a combination of model-based and model-free reinforcement learning [Dayan and Berridge, 2014], while deep learning systems can reach human-like performance in image classification [He et al., 2015], and speech recognition [Xiong et al., 2016] tasks. Combining these approaches may lead to algorithms that more closely resemble human learning, our best example of intelligence.

Recent successes in the field of reinforcement learning have been a result of combining standard reinforcement learning techniques that allow an agent to learn from experience, with modern deep learning techniques which allow the extraction of meaningful features from high dimensional data. However this combination presents some important challenges:

- Deep learning requires thousands or millions of samples, in reinforcement learning these sample must be generated by interacting with the world or an emulator and is time-consuming.
- The learning rates used in deep learning must be small to ensure training stability.
- The distribution that generates encountered samples is directly influenced by the policy being learned, meaning that the network may never recover from a bad update to the policy.

- The supervised learning algorithms currently used to train deep neural networks assume that all training examples are independent from each other and identically distributed (i.i.d.). In reinforcement learning the sequences of states that an agent encounters are strongly correlated.

I will now present the current state-of-the-art within deep reinforcement learning, and how these challenges are addressed.

2.3.1 Deep Q Learning

Mnih et al. [2013] introduced Deep Q-Networks (DQN), which represent a Q-function as a convolutional neural network. They are able to train a reinforcement learning agent that learns to play seven different Atari 2600 games directly from screen input. DQN uses the Q-learning algorithm presented in 2.1.2 with a few very important changes. DQN makes use of a *target* network with parameters θ^- to generate targets for Q function, and replaces the update in Equation (2.12) with

$$Q(s_t, a_t; \theta) \leftarrow Q(s_t, a_t; \theta) + \alpha\big[r_{t+i} + \gamma \max_{a'} Q(s_{t+1}, a'; \theta^-) - Q(s_t, a_t; \theta)\big] \tag{2.45}$$

The weights θ^- are update with the weights θ every few thousand updates, and kept fixed otherwise, which helps stabilize the learning algorithm.
They use experience replay, in which the 4-tuple representing agent's experiences (s_t, a_t, r_t, s_{s+1}) is stored in a database of a fixed size, referred to as a replay memory. The agent then learns by uniformly sampling experiences from the replay memory and applying the Q-learning update in Equation (2.45) to the sampled data. This presents several benefits over standard on-line Q-learning:

1. experiences are potentially used in more than one update
2. the correlation between samples uses to update the Q-network is broken
3. removing the feedback loops between policy updates and the distribution of encountered states
4. learning can be parallelized

Experience replay comes with a few drawbacks, it can only be used with off-policy learning algorithms that can update from data generated by a different policy, and it requires more memory and computation per experience.

The CNN used in DQN uses two convolutional layers, the first has 16 8×8 filters with a stride of 4 and applies a ReLU non-linearity, the second layer has 32

filters of size 4×4 and a stride of 2 with a ReLU non-linearity. The convolutional layers are followed by a fully connected layer with 256 hidden units and a ReLU non linearity, finally the output layer is a fully connected layer with one linear output for each valid action, as shown in Figure 2.9.
The input to the network is a $84 \times 84 \times 4$ tensor of the last four frames in the Atari emulator resized to 84×84 pixels and transformed to grayscale. DQN agents required 12-14 days of training on a powerful GPU.

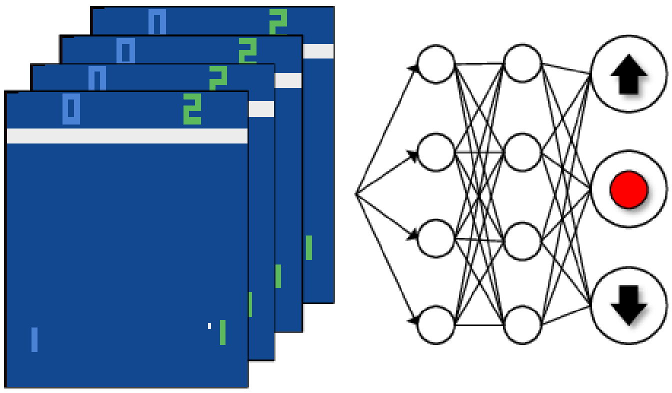

Figure 2.9: Illustration of the architecture used by Mnih et al. [2013].

Prioritized Experience Replay

Significant improvements over DQN can be achieved by sampling experiences from the replay memory in a non-uniform way. Schaul et al. [2015] assign a probability $P(i)$ to all samples in the replay memory, where

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{2.46}$$

for some constant α. The authors experiment with $p_i = |\delta_i| + \epsilon$ and $p_i = \frac{1}{\text{rank}(i)}$, where δ_i is the td error for that experience, and $\text{rank}(i)$ is the rank of that experience given by it's td error.
This technique significantly improves the sample efficientcy and performance of the algorithm in the Atari domain. To this date, some of the scores presented remain as the state-of-the-art.

General Reinforcement Learning Architecture

Nair et al. [2015] present the Gorila, a distributed architecture for deep reinforcement learning. The architecture is divided into four main components: multiple actors with their corresponding environment instances with each actor maintaining a replica of the centralized Q-network, a local per-actor replay memory and globally shared replay memory shared amongst all actors, a set of parameter servers that maintain a distributed representation of the Q-network, and multiple learner processes that sample experiences from the replay memory and update the parameters of the Q-network. Nair et al. [2015] follow the same model architecture presented in Mnih et al. [2015]. Their system is implemented using 100 actors distributed across 100 machines, in addition to 30 machines used as parameter servers. They are able to reach the performance of Mnih et al. [2015] in 38 games within 36 hours of training, and achieve their final performance after 4 days of training.

2.3.2 Deep Policy Gradient Algorithms

Policy gradient algorithms can also be adapted for use with deep learning models, and have been shown to outperform value based methods in some domains.

Deep Deterministic Policy Gradients

The notion of combining actor-critic algorithms with deep learning models was first explored in Lillicrap et al. [2015]. In this paper the authors present DDPG, an off-policy actor-critic algorithm featuring a deterministic policy, and deep neural networks to approximate the value function and policy. Following the techniques from Mnih et al. [2013], DDPG has slowly evolving target networks for both the value function and the policy which improves training stability, as well as a replay memory from which it samples experiences for learning. The critic is a Q function and is updated with the MSE of the Q learning target. The critic is updated with the deterministic policy gradient [Silver et al., 2014]

$$\nabla_\theta \eta = \mathbb{E}_{s\sim\pi_\theta}\left[\nabla_\theta \pi(s_t|\theta)\nabla_{a_t} Q(s_t, a_t)\right]. \tag{2.47}$$

DDPG achieves strong results in multiple continuous control tasks.

Asynchronous Advantage Actor-Critic

Mnih et al. [2016] combine deep neural networks with an on-policy advantage actor-critic algorithm with a stochastic policy. The actor network parameters are updated with RMSProp by following the direction of greatest ascent of the policy gradient, while the critic network parameters follow the direction of greatest descent given by the gradient of the MSE loss of the value function. For both the actor and the critic, the return is estimated with the n-step return given by

$$\widetilde{R}_t^{(n)} = \sum_{i=0}^{n-1} \gamma^i r_{t+1} + \gamma^n V(s_{t+n}; \theta_v). \tag{2.48}$$

The A3C algorithm is implemented with an asynchronous framework that shares many similarities with Nair et al. [2015], but is implemented on a single machine. Each actor is run on a separate CPU thread, and has access to its own environment instance, in addition to a copy of the model parameters. The framework maintains a central copy of the model parameters that the different actors can asynchronously update and synchronize from.

They show that the multiple parallel actors have a stabilizing effect on the training process, and are able to successfully train a one-step Q-learning agent (DQN), an n-step Q-learning agent, a one-step Sarsa agent without the need for experience replay.

A3C is able to achieve state-of-the-art performance on 19 Atari 2600 games by training for 4 days on a 16 core CPU. A3C is tested with two different CNN architectures. The first architecture is similar to DQN [Mnih et al., 2013], where the output layer for the Q-network is modified to output the policy and an output layer for the value function is added. The authors motivate the shared layers between the policy and the value network by noting that the features the network must learn in order to decide the best action, and to estimate the value of a state are probably similar. The second architecture experimented with in A3C includes a recurrent layer of 256 LSTM neurons placed between the final fully connected layer and the policy and value network layers. The recurrent architecture is able to outperform the feedforward architecture by a large margin in most games.

Parallel Advantage Actor-Critic

Clemente et al. [2017] introduce an advantage actor-critic algorithm that can be efficiently implemented on a GPU, allowing for reduced training times. The algorithm maintains a set of n_e environments and a singe neural network. Actions for all environments are queried simultaneously from the policy, and applied to all environments in parallel. The experience from all environments is gathered and used to calculate the gradient to update the network.

Actor-Critic with Experience Replay

Wang et al. [2016] introduce ACER, an off-policy advantage actor-critic algorithm based on A3C [Mnih et al., 2016] that is able to reuse old experiences by maintaining a replay memory, similar to the one used in in DQN [Mnih et al., 2013]. Targets are calculated in a similar fashion to A3C, but the return is estimated using the n-step off-policy algorithm Retrace [Munos et al., 2016]

$$Q^{\text{ret}}(s_t, a_t) = r_t + \gamma \min\left(c, \frac{\pi(a_{t+1}|s_{t+1})}{\beta(a_{t+1}|s_{t+1})}\right)\left[Q^{\text{ret}}(s_{t+1}, a_{t+1}) - Q(s_{t+1}, a_{t+1})\right] + \gamma V(s_{t+1}) \tag{2.49}$$

where c is a constant, $Q(s,a)$ is the current estimate of the Q function, $V(s) = \sum_{a'} Q(s,a')\pi(a'|s)$ and β is the policy that generated the samples. The Q function is then updated with Q^{ret} as a target, and the policy is updated with the gradient

$$\nabla_\theta \eta_{\pi_\theta} \approx \min\left(c', \frac{\pi(a_t|s_t)}{\beta(a_t|s_t)}\right)[Q^{\text{ret}}(s_t, a_t) - V(s_t)]\nabla_\theta \log \pi(a_t|s_t, \theta) \tag{2.50}$$

where c' is a constant.
The min used above limits the importance sampling ratio to the range $[0, c]$ which reduces the variance of the estimates, however they introduce a bias. This is addressed by using a bias correction term to the policy gradient.
ACER further constrains the policy gradient so that $D_{\text{KL}}[\pi(\cdot|s_t, \theta_a)||\pi(\cdot|s_t, \theta)] \leq \delta$ where δ is a constant, θ_a is the policy parameters after updating the policy using the policy gradient. D_{KL} is the Kullback-Leibler divergence given by

$$D_{\text{KL}}(P||Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)} \tag{2.51}$$

which is a measure of how one probability distribution diverges from another.
When tested on the Atari domain, ACER is able to outperform both A3C and DQN, while requiring significantly fewer samples.

Unsupervised Reinforcement and Auxiliary Learning

Jaderberg et al. [2016] is another A3C based algorithm that improves sample efficiency by storing experiences in a replay memory and reusing them a clever way. In addition to performing policy gradient learning on the policy and value function regression, UNREAL performs four additional learning tasks on the same data. All tasks share most of the parameters with the policy network, in addition to task-specific parameters.

The simplest task is *Value Function Replay*, in which a set of experiences are sampled from the replay memory and used to perform off-policy value function regression. The second task is *Reward Prediction*, where the network is given three consecutive frames and must predict whether the next frame will result in a positive, negative or zero reward. For this task the sampled experiences are skewed, so that rewarding and non-rewarding experiences are encountered with the same probability. The final two tasks are quite similar and only one of them is used for a given configuration. For *Pixel Control* each frame is divided into a 20×20 grid of non-overlapping 4×4 cells. Then 20×20 Q functions are trained where the instantaneous reward for that cell is given by the average absolute difference in value from the previous frame, where the average is taken over pixels and channels in the cell. The motivation of this task is for the network learn how its actions affect the world. *Feature Control* learns a Q function for each unit of a specific hidden layer, where the reward is given as a function of the activation of that unit. The motivation behind this task is to help the network learn the effect its actions have on its own activations.

Pixel Control alone greatly improves upon the sample efficiency and performance of A3C. Reward Prediction also gives a significant improvement, while value function replay results in only a modest improvement.

Chapter 3

Related Work

Similar approaches to the one I propose have been used for policy distillation and transfer learning. Policy distillation is when a dataset of state action pairs is available, and the task is to create a system that is able to emulate the observed behaviour and hopefully generate reasonable actions for states the system has never seen. Transfer learning is the task of using the knowledge a system has acquired when trained for a task to improve the training of the system on a different, but usually related, task.
Rusu et al. [2015] use a trained DQN [Mnih et al., 2015] network to generate a dataset $S^T = \{(s_i, \mathbf{q}_i)\}_{i=0}^N$ of state and q-value tuples. Then train a neural network with parameters θ with supervised learning on this dataset. The authors test three different objectives:

Minimizing the negative log-likelihood of the best action $a_i^* = \text{argmax}(\mathbf{q}_i)$

$$L_{NLL}(S^T, \theta) = -\sum_{i=1}^{|S^T|} \log P(a_i = a_i^* | s_i, \theta) \tag{3.1}$$

Minimizing the mean squared error with the q values

$$L_{MSE}(S^T, \theta) = \sum_{i=1}^{|S^T|} ||\mathbf{q}_i - \mathbf{Q}(\cdot, s_i|\theta)||_2^2 \tag{3.2}$$

Minimizing the KL divergence with temperature τ

$$L_{KL}(S^T, \theta) = \sum_{i=1}^{|S^T|} \text{softmax}(\frac{\mathbf{q}_i}{\tau}) \log \frac{\text{softmax}(\frac{\mathbf{q}_i}{\tau})}{\text{softmax}(\mathbf{Q}(\cdot, s_i|\theta))} \tag{3.3}$$

The KL divergence loss achieved the best performance closely followed by the negative log-likelihood loss and both matched or surpassed the performance of the original network. The mean squared error loss produced the worst performance.
The authors also try policy distillation during training, and are able to match the performance of DQN.

Parisotto et al. [2015] use a set of teacher DQN network trained on different tasks to generate a dataset $S^T = \{(s_i, \mathbf{q}_i, \mathbf{h}_i)\}_{i=0}^N$ where s_i and $\mathbf{q}_i$ are as in Rusu et al. [2015], and $\mathbf{h}_i$ is the vector of activations of a given layer in the teacher network. The authors then train two neural networks, a q network with paramters θ and a feature regression network with parameters θ_f.
The q network is trained to minimize the cross entropy

$$L_{CE}(S^T, \theta) = -\sum_{i=1}^{|S^T|} \text{softmax}(\mathbf{q}_i) \log\left(\text{softmax}(\mathbf{Q}(\cdot, s_i|\theta))\right) \tag{3.4}$$

and the feature regression network $f_i(\mathbf{h}(s_i|\theta); \theta_f)$ attempts to predict the teacher hidden states $\mathbf{h}_i$ from the activations $\mathbf{h}(s_i|\theta)$ of a given hidden layer of the q network. The feature regression network is trained to minimize the squared error

$$L_F(\theta, \theta_f) = ||f_i(\mathbf{h}(s_i|\theta); \theta_f) - \mathbf{h}_i||_2^2 \tag{3.5}$$

and the gradients are backpropagated back to the q network. These two losses encourage the q network to not only match the same policy as the teacher networks, but also to encode similar information as the teacher networks in its activations. Without the feature regression network, the q network is able to match the performance on 7 out of the 8 Atari games of the teacher networks using 10% of samples required to learn a single teacher network. The use of the feature regression network was shown to help in some games, but hinder learning in others.

Pritzel et al. [2017] present fully differentiable semi-tabular deep q learning agent that greatly increases sample efficiency. The agent uses the same convolutional neural network presented by Mnih et al. [2015] with the output layer removed. In addition, the agent maintains a dictionary D_i mapping keys h to q values q for each available action, where i is the index of the action.

Learning is quite different from other deep q learning algorithms. When a new state is encountered it is passed through the neural network to produce a key h. Then the q value for all actions is calculated by

$$Q_i = \sum_{q_j \in D_i} w_j q_j \tag{3.6}$$

where w_j is the weight given by

$$w_j = \frac{k(h, h_j)}{\sum\limits_{m} k(h, h_m)} \tag{3.7}$$

and $k(h, h_j)$ is an appropriate similarity metric between two keys. Meaning that the q value for a given state-action pair is the weighted average of all the q values of that action.

If a key h is already present in the dictionary, then its value is updated with the standard tabular q learning rule

$$Q_i \leftarrow Q_i + \alpha(Q^{(N)}(s, a) - Q_i) \tag{3.8}$$

where $Q^{(N)}(s, a)$ is the n-step estimate of the q value. If the key is not in the dictionary, then the key and value pair is simply appended to it. In order to make the agent scalable with an increasing dictionary, the similarity metric is computed the all elements of the dictionary and only the p most similar entries are used in Equation 3.6.

Chapter 4

Model

In this section a novel algorithm to combine reinforcement learning and deep learning is introduced.

Standard deep policy gradient methods represent the policy and the value functions as neural networks. Samples are gathered by interacting with the environment. Stochastic gradient ascent in used to train the policy network by following the direction given by the policy gradient theorem

$$\nabla_\theta \eta = R_t(s_t|\theta_v)\nabla_\theta \log \pi(a_t|s_t;\theta) \tag{4.1}$$

and the value network is trained with stochastic gradient descent by following the gradient of a

$$\nabla_{\theta_v} \eta = \nabla_{\theta_v} L_v(R_t(s_t|\theta_v), R_t^{\text{target}}) \tag{4.2}$$

I propose to divide the policy into two parts. First a neural network with parameters θ using the current state as input generates a parameter vector $f(s_t|\theta) = \phi_t$. Then, the policy is obtained by applying a deterministic function σ to ϕ_t so that $\pi(\cdot|s_t, \phi_t) = \sigma(\phi_t)$. With this parametrization, reinforcement learning can be used to find the vector of parameters

$$\phi'_t = \phi_t + \alpha_\phi \nabla_\phi \eta \tag{4.3}$$

that improves the current policy. Above, η is the return of the policy. Supervised learning in the form

$$\theta \leftarrow \theta - \alpha_\theta \nabla_\theta L(\phi_t, \phi'_t) \tag{4.4}$$

is used to train the neural network with ϕ'_t as its target.

Network targets ϕ' can be stored in a buffer and reused multiple times without importance sampling. However as the policy improves, the targets stored in the buffer no longer improve the current policy. Due to this, targets must be removed from the buffer after a certain number of timesteps.

This optimization procedure also allows α_ϕ to be much larger than α_θ since no gradients are backpropagated into the network when calculating ϕ', which can lead to a faster evolving policy. Additionally, expensive methods can be used to calculate $\nabla_\phi \eta$ since in general ϕ is several orders of magnitude smaller than θ.

This method can be motivated in multiple ways. It can be seen as policy distillation, where the teacher policy is the current policy after one step of SGD. Alternatively it can be thought of as a semi-tabular method, where the network targets ϕ' in the buffer are a tabular representation of the policy and supervised learning is performed on the network to match the tabular policy.

4.1 Parametrizing the policy

The naive way of parametrizing the policy would be to use a vector $\phi_t = \beta_s$, so that $\pi(a|s, \beta_s) = \beta_{s,j}$ where j is the index of the selected action. For convenience I will write β_s as β below.

The policy gradient with respect to β_s is then given by

$$\nabla_\beta \eta = A(a_t, s_t) \nabla_\beta \log(\pi(a_t|s_t, \beta)) \tag{4.5}$$
$$\nabla_\beta \eta = A(a_t, s_t) \nabla_\beta \log(\beta_j) \tag{4.6}$$
$$\nabla_\beta \eta = \frac{A(a_t, s_t)}{\beta_j} \nabla_\beta \beta_j \tag{4.7}$$

For index j 4.5 becomes (4.8)

$$\nabla_{\beta_j} \eta = \frac{A(a_t, s_t)}{\beta_j} \nabla_{\beta_j} \beta_j \tag{4.9}$$
$$\nabla_{\beta_j} \eta = \frac{A(a_t, s_t)}{\beta_j} \tag{4.10}$$
$$\nabla_{\beta_{i \neq j}} \eta = \frac{A(a_t, s_t)}{\beta_j} \nabla_{\beta_i} \beta_j \tag{4.11}$$
$$\nabla_{\beta_{i \neq j}} \eta = 0 \tag{4.12}$$

This can be written in vector notation as (4.13)

$$\nabla_\beta \eta = A(a_t, s_t) e_j \cdot \beta^{-1} \tag{4.14}$$

where

$$e_{j,l} = \begin{cases} 1, & \text{if } l = j \\ 0, & \text{otherwise} \end{cases}$$

and $[\beta^{-1}]_l = \frac{1}{\beta_l}$ for all $\beta_l \in \beta$.

The range of $\nabla_{\beta_j} \eta$ is $[-\infty, \infty]$. In addition, only the probability of the chosen action is updated. This means that using this estimate of the policy gradient to update the policy may change the policy infinitely, and even worse, result in a policy that is no longer a probability distribution.

The parametrization I have chosen to use, is to represent the policy as the softmax of a vector $\phi_t \in [-\infty, \infty]$, which I will write as ϕ, for each state so that

$$\pi(a_t|s_t, \phi) = \frac{e^{\phi_j}}{\sum_n e^{\phi_n}} \tag{4.15}$$
$$\pi(a_t|s_t, \phi) = \sigma(\phi)_j \tag{4.16}$$

The vector ϕ is referred to as the logits of the softmax. As I will later show, this parametrization results in a policy that is always a valid probability distribution, and gradients that are bound to a specific range.

4.2 Optimizing the Weights of the Neural Network

With the chosen parametrization in mind, multiple methods for optimizing θ may be devised. I will focus mainly on two methods; using the vector ϕ' directly as the target for the network, and using the target policy $\pi(\cdot|s_t, \phi') = \sigma(\phi')$ generated from the target vector as the network target.

4.2.1 Logit Target

The network weights can be updated to minimize the mean squared error between ϕ and ϕ'. However this is equivalent, within a constant factor, to simply performing gradient descent on the negative policy gradient. The proof of this can be seen in Appendix A. This method would be exactly the same as standard policy gradient methods used in deep reinforcement learning [Mnih et al. [2016], Wang et al. [2016], Clemente et al. [2017]].

4.2.2 Policy Target

The network weights can alternatively be updated to minimize the cross-entropy

$$\mathbb{H}(\pi(\cdot|s_t, \phi'), \pi(\cdot|s_t, \phi)) = -\sum_i \pi(a_i|s_t, \phi') \log \pi(a_i|s_t, \phi) \tag{4.17}$$

between the current policy $\pi(\cdot|s_t, \phi)$ and target policy $\pi(\cdot|s_t, \phi')$.

The standard approach is to use the policy gradient to update network parameters directly. This can result in infinitely large updates, while the use of this cross-entropy target results in gradients bounded to the range $[-1, 1]$.

The policy gradient with respect to the vector ϕ is then given by

$$\nabla_{\phi_t} \eta = A(a_t, s_t) \nabla_{\phi_t} \log(\sigma(\phi_t)_j) \tag{4.18}$$

By expanding the gradient we find

$$\nabla_\phi \eta = \frac{A(a_t, s_t)}{\sigma_j} \nabla_\phi \sigma(\phi_j) \tag{4.19}$$

$$\tag{4.20}$$

The derivative of the softmax function is given by

$$\nabla_\phi \sigma(\phi)_j = \sigma(\phi)_j (e_j - \sigma(\phi)) \tag{4.21}$$

the proof of this is shown in Appendix B.

$$\nabla_\phi \eta = \frac{A(a_t, s_t)}{\sigma(\phi_j)} \sigma(\phi_j)(e_j - \sigma(\phi)) \tag{4.22}$$
$$\nabla_\phi \eta = A(a_t, s_t)(e_j - \sigma(\phi)) \tag{4.23}$$
$$\tag{4.24}$$

By applying gradient ascent using the policy gradient we define

$$\phi' = \phi + \alpha \nabla_\phi \eta \tag{4.25}$$

We can then use ϕ' as a target policy given the current state, and train the neural network with the cross entropy loss given by $\mathbb{H}(\sigma', \sigma)$.

For brevity I will write $\sigma'(\phi)$ as σ' and $\sigma(\phi)$ as σ

$$\mathbb{H}(\sigma', \sigma) = -\sum_j \sigma'_j \log(\sigma_j) \tag{4.26}$$
$$\nabla_\phi \mathbb{H}(\sigma', \sigma) = -\nabla_\phi \sum_j \sigma'_j \log(\sigma_j) \tag{4.27}$$
$$\nabla_\phi \mathbb{H}(\sigma', \sigma) = -\sum_j \log(\sigma_j) \nabla_\phi \sigma'_j + \sigma'_j \nabla_\phi \log(\sigma_j) \tag{4.28}$$
$$\nabla_\phi \mathbb{H}(\sigma', \sigma) = -\sum_j \log(\sigma_j) \nabla_\phi \sigma'_j - \sum_j \sigma'_j \frac{\nabla_\phi \sigma_j}{\sigma_j} \tag{4.29}$$
$$\nabla_\phi \mathbb{H}(\sigma', \sigma) = -\sum_j \log(\sigma_j) \nabla_\phi \sigma'_j - \sum_j \sigma'_j \frac{\sigma_j (e_j - \sigma)}{\sigma_j} \tag{4.30}$$
$$\nabla_\phi \mathbb{H}(\sigma', \sigma) = -\sum_j \log(\sigma_j) \nabla_\phi \sigma'_j - \sum_j \sigma'_j e_j + \sigma \sum_j \sigma'_j \tag{4.31}$$
$$\nabla_\phi \mathbb{H}(\sigma', \sigma) = -\sum_j \log(\sigma_j) \nabla_\phi \sigma'_j + \sigma - \sigma' \tag{4.32}$$
$$\tag{4.33}$$

When calculating $\nabla_\phi \mathbb{H}(\sigma', \sigma)$, σ' is considered to be a constant relative to σ, from this follows that

$$\frac{\partial \phi'}{\partial \phi} = 0 \tag{4.34}$$

and therefore

$$\frac{\partial \sigma'}{\partial \phi} = 0 \tag{4.35}$$

By substituting this above, we can reduce the equation to

$$\nabla_\phi \mathbb{H}(\sigma', \sigma) = \sigma - \sigma' \tag{4.36}$$

The range of both σ' and σ is $[0, 1]$, from this it follows that

$$0 \leq \nabla_\phi \mathbb{H}(\sigma', \sigma) \leq 1 \tag{4.37}$$

The range of $A(a_t, s_t)$ is $[-\infty, \infty]$, and the range of σ_i is $[0, 1]$, therefore

$$-\infty \leq \nabla_\phi \eta \leq \infty \tag{4.38}$$

4.3 Proposed Algorithm

I propose a deep reinforcement learning algorithm that optimizes the weights of the neural network by using the target logits ϕ' directly. Algorithm 1 is a general algorithm, where any policy gradient estimate $\nabla_\phi \eta$ and any loss functions L and L_v could be used.

Algorithm 1

```
Initialize timestep counter $N = 0$, policy weights $\theta$ and value weights $\theta_v$
Initialize buffer $B$
Sample initial state $s_0$
for i $\in$ {0,1,...,T$_{\max}$} do
    Sample action $a_t$ from the policy $\pi(\cdot|s_t;\theta)$
    Perform action $a_t$ and observe state $s_{t+1}$ and reward $r_{t+1}$
    Calculate target $\phi'_t = \phi_t + \alpha_\phi \nabla_\phi \eta$
    Add $(s_t, \phi'_t)$ to buffer $B$
    Update $\theta \leftarrow \theta - \alpha_\pi \nabla_\theta L(\phi, \phi')$
    Update $\theta_v \leftarrow \theta_v - \alpha_v \nabla_{\theta_v} L_v(\theta_v)$
    for j $\in$ {0,1,...,n$_r$} do
        Sample M $(s, \phi')$ tuples from $B$
        Update $\theta \leftarrow \theta - \alpha_{\pi'} \frac{1}{M} \sum_{m=0}^{M} \nabla_\theta L(\phi_m, \phi'_m)$
    end for
end for
```

Chapter 5

Experiments

Throughout this section I will explore the performance of the proposed method with varying hyper-parameters, and compare it to other similar approaches.

5.1 Setup

All experiments are based on the open source PAAC [Clemente et al., 2017] implementation provided at `https://github.com/Alfredvc/paac`.
Experiments are run using TensorFlow version 1.0.1, and Python version 3.5.2.
The experiments will be run using the Arcade Learning Environment [Bellemare et al., 2013], an Atari 2600 emulator. A frame from the emulator is shown in

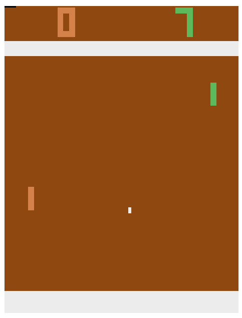

Figure 5.1: Sample frame from the game of Pong.

Figure 5.1. The game of Pong was chosen for all experiments. This is because this environment is significantly more challenging than other common benchmark

reinforcement learning benchmark, such as Cart Pole and Mountain Car, while only requiring a few hours of training.

5.2 Implementation

To test the proposed algorithm I will implement an advantage actor-critic algorithm based on PAAC [Clemente et al., 2017]. The policy and value networks are implemented as a single convolutional neural network, with a softmax output layer for the policy and a linear output layer for the value. The first layer of the network has 16 filters of size $8 \times 8 \times 4$ and a stride of 4. The second layer has 32 filters of size $4 \times 4 \times 16$ and a stride of 2. The policy layer has 4 units, one per action. And the value layer has a single unit.
Following the procedure of Clemente et al. [2017] each action is repeated 4 times, and the per-pixel maximum value from the two last frames is used as the input to the neural network. The emulator provides frames of size 210×160 and 3 color channels. The frames are rescaled to 80×80 pixels and the colors are transformed to grayscale.

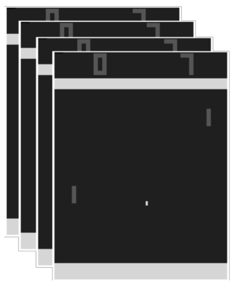
Figure 5.2: Sample input to the network.

The value function parameters θ_v are updated with stochastic gradient decent, following the gradient given by

$$\nabla_{\theta_v} L_v(\theta_v) = \nabla_{\theta_v} \left(Q^{(5)}(s_t, a_t; \theta, \theta_v) - V(s_t; \theta_v)\right)^2 \tag{5.1}$$

where $Q^{(5)}$ is the 5-step estimate of the Q function, and the policy parameters are updated by stochastic gradient descent, following the gradient given by

$$\nabla_\theta L(\phi_t, \phi_t') = \mathbb{H}(\pi(\cdot|s_t, \phi'), \pi(\cdot|s_t, \phi)) \tag{5.2}$$

5.2.1 Experience Replay

The algorithm I propose reuses information gathered from previous experiences to aid learning and increase sample efficiency. In reinforcement learning, this is referred to as experience replay. In order to compare my method to standard experience replay, I implement an off-policy version of PAAC [Clemente et al., 2017]. Experiences are used for on-policy updates when they are experienced, and then stored in a replay memory to be reused in the future.

This novel implementation takes inspiration from the work of Wang et al. [2016], using truncated importance sampling and Retrace [Munos et al., 2016] estimates of the value function. Below, π is the current policy and β is the policy that generated the sample. The truncated importance sampling ratio is given by

$$p_t = \min\left(1, \frac{\pi(a_t|s_t;\theta)}{\beta(a_t|s_t)}\right) \tag{5.3}$$

and the off-policy n-step estimate of the value function is given by

$$V^{(\mathrm{N})}(s_t) = \sum_{\tau=t}^{t+N} r_\tau \gamma^{\tau-t} \prod_{i=t}^{\tau} p_i + \gamma^N V(s_{t+N};\theta_v) \tag{5.4}$$

Finally, the policy is updated in the direction of greatest ascent given by the truncated off-policy policy gradient

$$\nabla_\theta \eta = \min\left(10, \frac{\pi(a_t|s_t;\theta)}{\beta(a_t|s_t)}\right) \nabla_\theta \log \pi(a_t|s_t;\theta)[V^{(\mathrm{N})}(s_t) - V(s_t;\theta_v)] \tag{5.5}$$

5.3 Results

In this section I start by evaluating the proposed algorithm's performance, then the effect of the hyperparameters introduced in the algorithm, and finally compare its performance to the PAAC algorithm. All tests are performed on the game of Pong.

First I performed a grid search across multiple orders of magnitude to find an appropriate learning rate for the policy loss α_π, and the learning rate of the targets α_ϕ. The learning rate for the value network α_v is kept the same as in Clemente et al. [2017].

The best performing hyperparameters were $\alpha_\pi = 0.1$ and $\alpha_\phi = 1.0$. During the hyperparameter search, it was discovered that different learning rates must be used for updates with on-policy and off-policy data. This introduced a new hyperparameter $\alpha_{\pi'}$ that is the learning rate for the policy loss when using targets from the replay memory. The best value found for this hyperparameter was 0.01. The average of 3 runs using these hyperparameters can be seen in Figure 5.3.

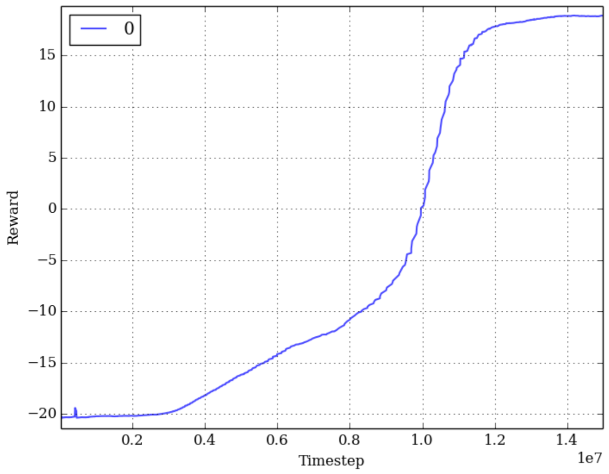

Figure 5.3: Average performance of three runs with the best found hyperparameters.

Targets are reused by sampling (s, ϕ') tuples from the the replay memory, and updating the network with the sampled tuples. This is done n_r times per iteration of the algorhtm. To understand the usefulness of reusing targets, tests were run with $n_r \in \{0, 1, 2, 4, 8, 16\}$. As can be shown in Figure 5.4 more off-policy updates reduces the amount of timesteps needed for reach a given average rewards, however doing this more than once per iteration results in diminishing returns. The best sample efficiency is observed when performing 16 off-policy updates per iteration.

As mentioned before, old targets must be thrown out after some time. To examine how long targets could be kept, tests were run where targets were kept for 10, 100, 1000, and 10000 timesteps and a single off-policy update was performed per iteration. The tests shown in Figure 5.5 show that targets should be removed from the replay memory after 100 timesteps.

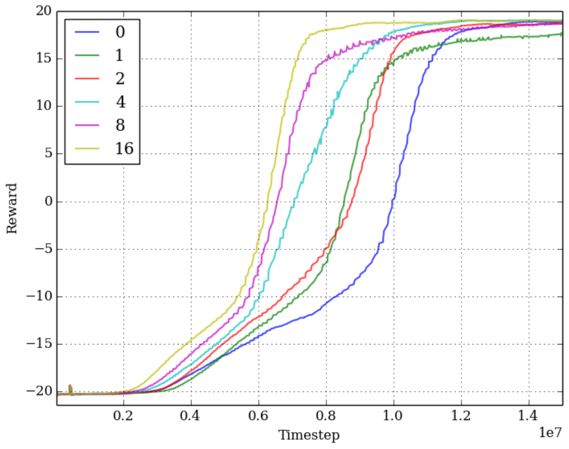

Figure 5.4: Average performance of three runs, using different amount of target reuse.

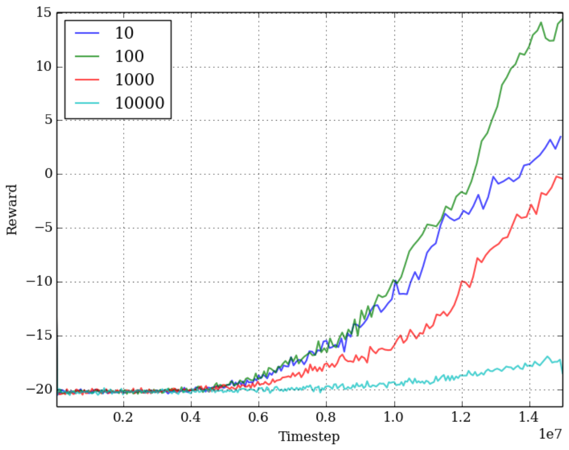

Figure 5.5: Performance keeping targets for different amounts of timesteps

5.3.1 PAAC Baseline

To evaluate how the proposed algorithm compares to current state of the art algorithms, it is necessary to have a baseline algorithm to compare it to. Given that the proposed algorithm's implementation was based on PAAC [Clemente et al., 2017], it was also used as a baseline. Tests were run on the game of Pong with the standard hyperparameters presented in PAAC. Three tests were run with this setup, the results are shown together with the proposed algorithm without target reuse in Figure 5.6. Both algorithms show very similar performance, with the proposed algorithm performing slightly better.

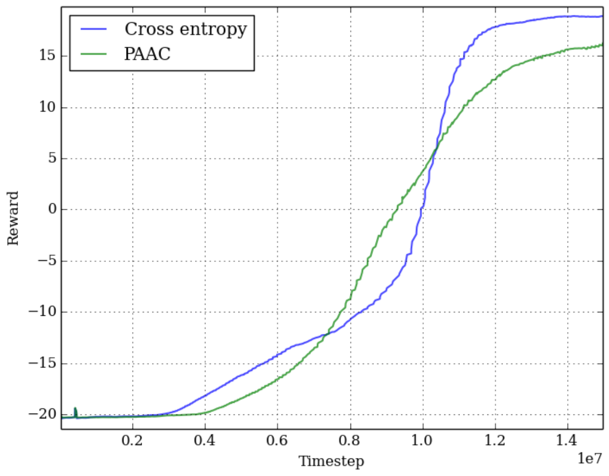

Figure 5.6: Performance of the proposed algorithm and PAAC on the game of Pong.

One of the benefits of the proposed algorithm, is that it does not require importance sampling in order to reuse old experiences. Standard off-policy reinforcement learning algorithms require importance sampling, since experiences are aquired using a policy that is no longer the current policy. Importance sampling can lead to instabilities, and techniques such as Retrace [Munos et al., 2016] and truncation [Wang et al., 2016] must be used. To evaluate the effect of importance sampling in standard reinforcement learning algorithms tests were run on my off-policy PAAC implementation with different replay memory sizes, both with and without importance sampling.

Figure 5.7 shows the performance of off-policy PAAC with and without importance sampling. It is clear from these tests that importance sampling is essential for the convergence of the algorithm. The best performance is achieved when storing targets for over 1000 timesteps. PAAC is able to learn without importance sampling when experiences are only kept for up to 100 timesteps, and catastrophic divergence if they are kept for any longer. This is because the policy changes slowly, therefore the current policy is still similar to the behavioural policy.

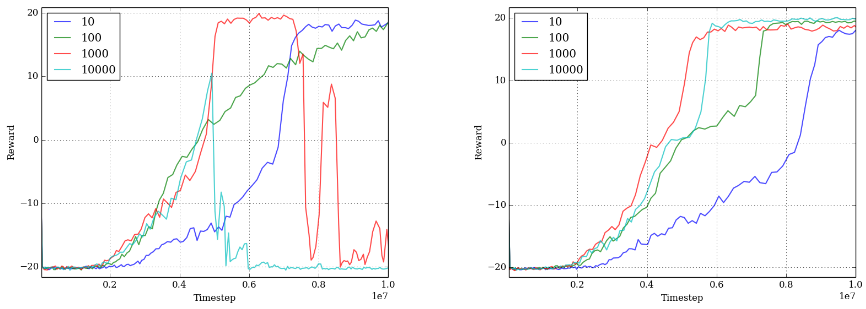

Figure 5.7: Performance of off-policy PAAC without (left), and with (right) importance sampling, for multiple replay memory sizes..

Chapter 6

Discussion

In this section we explore the results of the experiments, and the importance of these results. At the time of writing PAAC [Clemente et al., 2017], it was a state of the art algorithm in the Atari domain. Both the proposed algorithm and the off-policy version of PAAC implemented in this thesis offer between two and three times the sample efficiency of the original PAAC algorithm.

It is clear from the results of the performed experiments that without reusing targets the proposed algorithm can match and even surpass the performance of on-policy PAAC. Off-policy PAAC does show better performance than the proposed algorithm, even when reusing targets 16 times per iteration. While it may seem that this indicates that off-policy PAAC is superior, that is not the case. The target reuse in the proposed algorithm is conceptually different from the off-policy updates performed with off-policy PAAC. When targets are generated, they specify a point in policy space that is better than the current location of the policy in policy space. In other terms, a target specifies a distribution over actions for the current state that would lead to a higher return than the current distribution over actions dictated by the current policy. When an update is performed, the current policy is moved in a direction that is closer to the target, and no changes are made to the target. The off-policy PAAC samples the raw experiences, represented as $(s_t, a_t, \pi(a_t|s_t), s_{t+1}, r_{t+1})$ 5-tuples and from this directly calculates a gradient that would steer the policy so that it leads to a higher return after performing an update. This can be thought of as re-calculating the targets on every update. The proposed algorithm is able to reuse targets without re-calculating them.

Target reuse and off-policy updates can be combined to increase sample efficiency even further. Instead of only storing state-target tuples in the buffer, the tuple representing the experience can be stored alongside the state-target tuple. Once a target becomes stale, instead of removing it from the buffer the targets can be generated once again with for example an off-policy estimate of the policy gradient. The old state-target tuple can then be updated with the new target, and used for updates until it becomes stale again.

The PAAC baseline uses standard policy gradient updates, which may result in gradients that are infinitely large. Due to this, gradient clipping must be applied to the gradients before updating the network parameters. Gradient clipping reduces the magnitude of the gradient across all its dimensions so that it is below some threshold. This does lead to gradients that are bounded, however information is lost in the process. Given that the magnitude of a vector is given by

$$||v||_2 = \sqrt{\sum_i v_i^2} \tag{6.1}$$

even if only a single dimension of the gradient is excessively large, all the dimensions of gradient must be scaled down by some factor. This means that after the clipping is performed, almost all the information in the gradient will be lost, except for the largest dimension. The proposed algorithm avoids this issue completely, as the gradients are guaranteed to be within the range $[-1, 1]$.

Chapter 7

Conclusion and Future Work

The proposed algorithm explores a novel way of combining reinforcement learning algorithms with the powerful function approximation offered by deep learning. Reinforcement learning is used to calculate policy targets that lead to better returns, and then standard supervised learning methods can be used to train a neural network to match the targets. The proposed algorithm is able to match the performance of PAAC in the game of Pong, while requiring fewer interactions with the environment.

Supervised learning in neural networks is well understood, there exists a multitude of tools to aid in their training, and an abundance of documentation on best practices and techniques to build successful models. Deep reinforcement learning algorithms on the other hand are a relative new invention, they are usually complex, and difficult to reason about. By decoupling the reinforcement learning and deep learning components of these algorithms, it is possible to reason about them in isolation.

This separation also provides a clean interface between the two components, the policy targets. This allows for great flexibility when choosing both the reinforcement learning and the deep learning algorithm. Multiple reinforcement learning algorithms could be used simultaneously under training to generate the targets, something that would be difficult to do with standard deep reinforcement learning algorithms.

This general algorithm could also be used in partially observable domains, where we wish to train an agent to perform well in an environment in which it does not have access to the whole state. For example in the game of poker. A reinforcement learning algorithm can be allowed full access to the state, in the case of poker access to the other player's cards, and use this information to generate optimal targets. A neural network can then be trained using a limited state information, in the case of poker only the agent's cards and the board, to match the targets provided by the reinforcement learning algorithm with full state access. The resulting algorithm is able to use knowledge of the full state to learn, but act only using the partial state it as access to.

Appendices

Appendix A

The proof that using a mean squared error with logit targets is equivalent, within a constant factor, to using the policy gradient directly is as follows

$$\mathrm{MSE}(\phi, \phi') = \frac{1}{N}\sum_n (\phi_n - \phi'_n)^2 \tag{A.1}$$

$$\nabla_\phi \mathrm{MSE}(\phi, \phi') = \frac{1}{N}\sum_n \nabla_\phi (\phi_n - \phi'_n)^2 \tag{A.2}$$

$$\nabla_\phi \mathrm{MSE}(\phi, \phi') = \frac{2}{N}\sum_n (\phi_n - \phi'_n)\nabla_\phi(\phi_n - \phi'_n) \tag{A.3}$$

$$\tag{A.4}$$

from Equation 4.25 we have that

$$\phi' = \phi + \nabla_\phi \eta \tag{A.5}$$

$$\nabla_\phi \mathrm{MSE}(\phi, \phi') = \frac{2}{N}\sum_n (\phi_n - \phi_n - \alpha\nabla_\phi \eta)\nabla_\phi \phi_n \tag{A.6}$$

$$\nabla_\phi \mathrm{MSE}(\phi, \phi') = \frac{-2\alpha\nabla_\phi \eta}{N}\sum_n e_n \tag{A.7}$$

$$\tag{A.8}$$

where

$$e_{j,l} = \begin{cases} 1, & \text{if } l = j \\ 0, & \text{otherwise} \end{cases}$$

$$\nabla_\phi \mathrm{MSE}(\phi, \phi') = \frac{-2\alpha}{N}\nabla_\phi \eta \tag{A.9}$$

$$\tag{A.10}$$

Appendix B

The derivative of the softmax function can be derived as follows

$$\sigma(\phi)_j = \frac{e^{\phi_j}}{\sum_n e^{\phi_n}} \tag{B.1}$$

$$\nabla_\phi \sigma(\phi)_j = \nabla_\phi \left(e^{\phi_j} (\sum_n e^{\phi_n})^{-1} \right) \tag{B.2}$$

$$\nabla_\phi \sigma(\phi)_j = (\sum_n e^{\phi_n})^{-1} \nabla_\phi e^{\phi_j} + e^{\phi_j} \nabla_\phi (\sum_n e^{\phi_n})^{-1} \tag{B.3}$$

$$\nabla_\phi \sigma(\phi)_j = \frac{\nabla_\phi e^{\phi_j}}{\sum_n e^{\phi_n}} - \frac{e^{\phi_j} \sum_n \nabla_\phi e^{\phi_n}}{(\sum_n e^{\phi_n})^{-2}} \tag{B.4}$$

$$\tag{B.5}$$

the jth component of the gradient is given by

$$\frac{\partial \sigma(\phi)_j}{\partial \phi_j} = \frac{e^{\phi_j}}{\sum_n e^{\phi_n}} - \frac{e^{\phi_j} e^{\phi_j}}{(\sum_n e^{\phi_n})^{-2}} \tag{B.6}$$

$$\tag{B.7}$$

using the definition of the softmax function

$$\frac{\partial \sigma(\phi)_j}{\partial \phi_j} = \sigma(\phi)_j (1 - \sigma(\phi)_j) \tag{B.8}$$

$$\frac{\partial \sigma(\phi)_j}{\partial \phi_{i \neq j}} = -\frac{e^{\phi_j} e^{\phi_i}}{(\sum_n e^{\phi_n})^{-2}} \tag{B.9}$$

$$\frac{\partial \sigma(\phi)_j}{\partial \phi_{i \neq j}} = -\sigma(\phi)_j \sigma(\phi_i) \tag{B.10}$$

$$\tag{B.11}$$

this can be written in vector notation as

$$\nabla_\phi \sigma(\phi)_j = \sigma(\phi)_j (e_j - \sigma(\phi)) \tag{B.12}$$

where

$$e_{j,l} = \begin{cases} 1, & \text{if } l = j \\ 0, & \text{otherwise} \end{cases}$$

Bibliography

Baird III, L. C. (1993). Advantage updating. Technical report, DTIC Document.

Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279.

Bottou, L., Curtis, F. E., and Nocedal, J. (2016). Optimization Methods for Large-Scale Machine Learning. *ArXiv e-prints*.

Clemente, A. V., Castejón, H. N., and Chandra, A. (2017). Efficient Parallel Methods for Deep Reinforcement Learning. *ArXiv e-prints*.

Dayan, P. and Berridge, K. C. (2014). Model-based and model-free pavlovian reward learning: revaluation, revision, and revelation. *Cognitive, Affective, & Behavioral Neuroscience*, 14(2):473–492.

Degris, T., White, M., and Sutton, R. S. (2012). Off-Policy Actor-Critic. *ArXiv e-prints*.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). Deep learning. Book in preparation for MIT Press.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep Residual Learning for Image Recognition. *ArXiv e-prints*.

Jaderberg, M., Mnih, V., Czarnecki, W. M., Schaul, T., Leibo, J. Z., Silver, D., and Kavukcuoglu, K. (2016). Reinforcement Learning with Unsupervised Auxiliary Tasks. *ArXiv e-prints*.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc.

Lample, G. and Singh Chaplot, D. (2016). Playing FPS Games with Deep Reinforcement Learning. *ArXiv e-prints*.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2015). Continuous control with deep reinforcement learning. *ArXiv e-prints*.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning. *ArXiv e-prints*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

Mnih, V., Puigdomènech Badia, A., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous Methods for Deep Reinforcement Learning. *ArXiv e-prints*.

Munos, R., Stepleton, T., Harutyunyan, A., and Bellemare, M. G. (2016). Safe and Efficient Off-Policy Reinforcement Learning. *ArXiv e-prints*.

Nair, A., Srinivasan, P., Blackwell, S., Alcicek, C., Fearon, R., De Maria, A., Panneershelvam, V., Suleyman, M., Beattie, C., Petersen, S., Legg, S., Mnih, V., Kavukcuoglu, K., and Silver, D. (2015). Massively Parallel Methods for Deep Reinforcement Learning. *ArXiv e-prints*.

Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814.

Parisotto, E., Ba, J. L., and Salakhutdinov, R. (2015). Actor-mimic: Deep multitask and transfer reinforcement learning. *arXiv preprint arXiv:1511.06342*.

Pritzel, A., Uria, B., Srinivasan, S., Puigdomènech, A., Vinyals, O., Hassabis, D., Wierstra, D., and Blundell, C. (2017). Neural Episodic Control. *ArXiv e-prints*.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. (2014). ImageNet Large Scale Visual Recognition Challenge. *ArXiv e-prints*.

Rusu, A. A., Gomez Colmenarejo, S., Gulcehre, C., Desjardins, G., Kirkpatrick, J., Pascanu, R., Mnih, V., Kavukcuoglu, K., and Hadsell, R. (2015). Policy Distillation. *ArXiv e-prints*.

Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2015). Prioritized Experience Replay. *ArXiv e-prints.*

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489.

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., and Riedmiller, M. (2014). Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 387–395.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge.

Sutton, R. S., McAllester, D. A., Singh, S. P., Mansour, Y., et al. (1999). Policy gradient methods for reinforcement learning with function approximation. In *NIPS*, volume 99, pages 1057–1063.

Wang, Z., Bapst, V., Heess, N., Mnih, V., Munos, R., Kavukcuoglu, K., and de Freitas, N. (2016). Sample Efficient Actor-Critic with Experience Replay. *ArXiv e-prints.*

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256.

Wilson, D. R. and Martinez, T. R. (2003). The general inefficiency of batch training for gradient descent learning. *Neural Networks*, 16(10):1429–1451.

Xiong, W., Droppo, J., Huang, X., Seide, F., Seltzer, M., Stolcke, A., Yu, D., and Zweig, G. (2016). Achieving Human Parity in Conversational Speech Recognition. *ArXiv e-prints.*